Concurrency in Go

Feb 2\textsuperscript{nd} & 3\textsuperscript{rd} 2022
Go Resources

https://tour.golang.org/list
https://play.golang.org
https://gobyexample.com
Outline

Two Synchronization Mechanisms
Locks
Channels
MapReduce

A Case Study of WordCount
Two synchronization mechanisms

**Locks** - limit access to a critical section

**Channels** - pass information across processes using a queue
Example: Bank Account

Bob

Read b = 100

b = b + 10

Write b = 110

Alice

Read b = 110

b = b + 10

Write b = 120
Example: Bank Account

Bob

- Read $b = 100$
- $b = b + 10$
- Write $b = 110$

Alice

- Read $b = 100$
- $b = b + 10$
- Write $b = 110$ (highlighted in red)
What went wrong?

Changes to balance are not \textit{atomic}

```python
func Deposit(amount) {
    lock balanceLock
    read balance
    balance = balance + amount
    write balance
    unlock balanceLock
}
```
package account

import "sync"

type Account struct {
    balance int
    lock sync.Mutex
}

func NewAccount(init int) Account {
    return Account{balance: init}
}

func (a *Account) Deposit(v int) {
    a.lock.Lock()
    defer a.lock.Unlock()
    a.balance += v
}

func (a *Account) CheckBalance() int {
    a.lock.Lock()
    defer a.lock.Unlock()
    return a.balance
}

func (a *Account) Withdraw(v int) {
    a.lock.Lock()
    defer a.lock.Unlock()
    a.balance -= v
}

func (a *Account) Deposit(v int) {
    a.lock.Lock()
    defer a.lock.Unlock()
    a.balance += v
}
package account

import "sync"

type Account struct {
    balance int
    lock sync.RWMutex
}

func NewAccount(init int) Account {
    return Account{balance: init}
}

func (a *Account) Deposit(v int) {
    a.lock.Lock()
    defer a.lock.Unlock()
    a.balance += v
}

func (a *Account) CheckBalance() int {
    a.lock.RLock()
    defer a.lock.RUnlock()
    return a.balance
}

func (a *Account) Withdraw(v int) {
    a.lock.Lock()
    defer a.lock.Unlock()
    a.balance -= v
}

func (a *Account) Deposit(v int) {
    a.lock.Lock()
    defer a.lock.Unlock()
    a.balance += v
}
Two Solutions to the Same Problem

Locks:
Multiple threads can reference same memory location
Use lock to ensure only one thread is updating it at any given time

Channels:
Data item initially stored in channel
Threads must request item from channel, make updates, and return item to channel
Go channels

Channels also allow us to safely communicate between goroutines.

```go
// Launch workers
for i := 0; i < numWorkers; i++ {
    go func() {
        // ... do some work
        result <- i
    }()
}

// Wait until all worker threads have finished
for i := 0; i < numWorkers; i++ {
    handleResult(<-result)
}
fmt.Println("Done!")
```
Go channels

Easy to express
asynchronous RPC

```go
result := make(chan int, numServers)

// Send query to all servers
for i := 0; i < numServers; i++ {
    go func() {
        resp := // ... send RPC to server
        result <- resp
    }()
}

// Return as soon as the first server responds
handleResponse(<-result)
```
package account

type Account struct {
    // Fill in Here
}

func NewAccount(init int) Account {
    // Fill in Here
}

func (a *Account) CheckBalance() int {
    // What goes Here?
    }

func (a *Account) Withdraw(v int) {
    // ???
    }

func (a *Account) Deposit(v int) {
    // ???
    }
package account

type Account struct {
    balance chan int
}

func NewAccount(init int) Account {
    a := Account{make(chan int, 1)}
    a.balance <- init
    return a
}

func (a *Account) CheckBalance() int {
    // What goes Here?
}

func (a *Account) Withdraw(v int) {
    // ???
}

func (a *Account) Deposit(v int) {
    // ???
}
package account

type Account struct {
    balance chan int
}

func NewAccount(init int) Account {
    a := Account{make(chan int, 1)}
    a.balance <- init
    return a
}

func (a *Account) CheckBalance() int {
    bal := <-a.balance
    a.balance <- bal
    return bal
}

func (a *Account) Withdraw(v int) {
    // ????
}

func (a *Account) Deposit(v int) {
    // ????
}
package account

type Account struct {
    balance chan int
}

func NewAccount(init int) Account {
    a := Account{make(chan int, 1)}
    a.balance <- init
    return a
}

func (a *Account) CheckBalance() int {
    bal := <-a.balance
    a.balance <- bal
    return bal
}

func (a *Account) Withdraw(v int) {
    bal := <-a.balance
    a.balance <- (bal - v)
}

func (a *Account) Deposit(v int) {
    //???
}
package account

type Account struct {
    balance chan int
}

func NewAccount(init int) Account {
    a := Account{make(chan int, 1)}
    a.balance <- init
    return a
}

func (a *Account) CheckBalance() int {
    bal := <-a.balance
    a.balance <- bal
    return bal
}

func (a *Account) Withdraw(v int) {
    bal := <-a.balance
    a.balance <- (bal - v)
}

func (a *Account) Deposit(v int) {
    bal := <-a.balance
    a.balance <- (bal + v)
}
Select statement

`select` allows a goroutine to wait on multiple channels at once

```go
for {
    select {
        case money := <-dad:
            buySnacks(money)
        case money := <-mom:
            buySnacks(money)
    }
}
```
Select statement

`select` allows a goroutine to wait on multiple channels at once

```go
for {
    select {
        case money := <-dad:
            buySnacks(money)
        case money := <-mom:
            buySnacks(money)
        case default:
            starve()
            time.Sleep(5 * time.Second)
    }
}
```
Handle timeouts using select

```go
result := make(chan int)
timeout := make(chan bool)

// Asynchronously request an answer from server, timing out after X seconds
askServer(result, timeout)

// Wait on both channels
select {
    case res := <-result:
        handleResult(res)
    case <-timeout:
        fmt.Println("Timeout!")
}

func askServer(result chan int, timeout chan bool) {
    // Start timer
    go func() {
        time.Sleep(5 * time.Second)
        timeout <- true
    }()
    // Ask server
    go func() {
        response := // ... send RPC
        result <- response
    }()
}
```
Handle timeouts using `select`

```go
define variables and functions
    result := make(chan int)
    timeout := make(chan bool)
    askServer(result, timeout)

    // Wait on both channels
    select {
        case res := <-result:
            handleResult(res)
        case <-timeout:
            fmt.Println("Timeout!")
    }

    func askServer(
        result chan int,
        timeout chan bool) {
        // Start timer
        go func() {
            time.Sleep(5 * time.Second)
            timeout <- true
        }()

        // Ask server
        go func() {
            response := // ... send RPC
            result <- response
        }()
```
Exercise: Implementing a mutex using channels

type Lock struct {
// ???
}

func NewLock() Lock {
// ???
}

func (l *Lock) Lock() {
// ???
}

func (l *Lock) Unlock() {
// ???
}
Exercise: Implementing a mutex using channels

```go
type Lock struct {
    ch chan bool
}

func NewLock() Lock {
    // ???
}

func (l *Lock) Lock() {
    // ???
}

func (l *Lock) Unlock() {
    // ???
}
```
Exercise: Implementing a mutex using channels

type Lock struct {
    ch chan bool
}

func NewLock() Lock {
    l := Lock{make(chan bool, 1)}
    l.ch <- true
    return l
}

func (l *Lock) Lock() {
    // ???
}

func (l *Lock) Unlock() {
    // ???
}
Exercise: Implementing a mutex using channels

type Lock struct {
    ch chan bool
}

func NewLock() Lock {
    l := Lock{make(chan bool, 1)}
    l.ch <- true
    return l
}

func (l *Lock) Lock() {
    <-l.ch
}

func (l *Lock) Unlock() {
    // ???
}
Exercise: Implementing a mutex using channels

type Lock struct {
    ch chan bool
}

func NewLock() Lock {
    l := Lock{make(chan bool, 1)}
    l.ch <- true
    return l
}

func (l *Lock) Lock() {
    <-l.ch
}

func (l *Lock) Unlock() {
    l.ch <- true
}
Outline

Two synchronization mechanisms

Locks

Channels

MapReduce

A Case Study of WordCount
Application: WordCount

How much wood would a woodchuck chuck if a woodchuck could chuck wood?

how: 1, much: 1, wood: 2, would: 1, a: 2, woodchuck: 2, chuck: 2, if: 1, could: 1
Application: WordCount

**Locally**: Tokenize and store words in a hash map

**How do you parallelize this?**

Split document by half

Build two hash maps, one for each half
How do you do this in a distributed environment?
When in the Course of human events, it becomes necessary for one people to dissolve the political bands which have connected them with another, and to assume, among the Powers of the earth, the separate and equal station to which the Laws of Nature and of Nature's God entitle them, a decent respect to the opinions of mankind requires that they should declare the causes which impel them to the separation.
When in the Course of human events, it becomes necessary for one people to dissolve the political bands which have connected them with another, and to assume, among the Powers of the earth, the separate and equal station to which the Laws of Nature and of Nature's God entitle them, a decent respect to the opinions of mankind requires that they should declare the causes which impel them to the separation.

Partition
When in the Course of human events, it becomes necessary for one people to dissolve the political bands which have connected them with another, and to assume, among the Powers of the earth, the separate and equal station to which the Laws of Nature and of Nature’s God entitle them, a decent respect to the opinions of mankind requires that they should declare the causes which impel them to the separation.
when: 1, in: 1,
the: 1, course: 1,
of: 1, human: 1,
events: 1, it: 1

dissolve: 1, the: 2,
political: 1, bands: 1,
which: 1, have: 1,
connected: 1, them: 1

nature: 2, and: 1, of: 2,
god: 1, entitle: 1, them: 1,
decent: 1, respect: 1,
mankind: 1, opinion: 1

requires: 1, that: 1,
they: 1, should: 1,
declare: 1, the: 1,
causes: 1, which: 1

among: 1, the: 2,
powers: 1, of: 2, earth: 1, separate: 1, equal: 1, and: 1
when: 1, in: 1, the: 1, course: 1, of: 1, human: 1, events: 1, it: 1
dissolve: 1, the: 2, political: 1, bands: 1, which: 1 ...
among: 1, the: 2, powers: 1, of: 2, earth: 1, separate: 1, equal: 1, and: 1 ...
nature: 2, and: 1, of: 2, god: 1, entitle: 1, them: 1, decent: 1, respect: 1, mankind: 1, opinion: 1 ...
requires: 1, that: 1, they: 1, should: 1, declare: 1, the: 1, causes: 1, which: 1 ...
Now … How to merge results?
Compute word counts locally
Merging results computed locally

Several options

Don’t merge — requires additional computation for correct results

Send everything to one node — what if data is too big? Too slow...

Partition key space among nodes in cluster (e.g. [a-e], [f-j], [k-p] ...)

1. Assign a key space to each node
2. Split local results by the key spaces
3.
when: 1, in: 1, the: 1, course: 1, of: 1, human: 1, events: 1, it: 1, dissolve: 1, the: 2, political: 1, bands: 1, which: 1, have: 1, connected: 1, them: 1 ... requires: 1, that: 1, they: 1, should: 1, declare: 1, the: 1, causes: 1, which: 1 ... among: 1, the: 2, powers: 1, of: 2, earth: 1, separate: 1, equal: 1, and: 1 ... nature: 2, and: 1, of: 2, god: 1, entitle: 1, them: 1, decent: 1, respect: 1, mankind: 1, opinion: 1 ...
when: 1, the: 1, in: 1, it: 1, human: 1, course: 1, events: 1, of: 1

causes: 1, declare: 1, requires: 1, should: 1, that: 1, they: 1, the: 1, which: 1

bands: 1, dissolve: 1, connected: 1, have: 1, political: 1, the: 1, them: 1, which: 1

nature: 2, of: 2, mankind: 1, opinion: 1, entitle: 1, and: 1, decent: 1, god: 1, them: 1, respect: 1, among: 1, and: 1, equal: 1, earth: 1, separate: 1, the: 2, powers: 1, of: 2

Split local results by key space
All-to-all shuffle
MapReduce

Partition dataset into many chunks

**Map stage:** Each node processes one or more chunks locally

**Reduce stage:**
MapReduce Interface

\texttt{map(key, value) -> list(<k', v'>)}

Apply function to (key, value) pair
Outputs list of intermediate pairs

\texttt{reduce(key, list<value>) -> <k', v'>}
Applies aggregation function to values
MapReduce: WordCount

map(key, value):
// key = document name
// value = document contents
    for each word w in value:
        emit (w, 1)

reduce(key, values):
// key = the word
// values = number of occurrences of that word
count = sum(values)
emit (key, count)
MapReduce: WordCount

MapReduce algorithm flow:

1. **Map**:
   - Takes input data and applies a function to each key-value pair.
   - Outputs intermediate key-value pairs.
   - Example:
     - Input: "How much wood would a woodchuck chuck if a woodchuck could chuck wood?"
     - Output: (word, frequency)

2. **Shuffle**:
   - Sorts the intermediate key-value pairs so that values are grouped together.
   - Example:
     - Input: (word, frequency) pairs
     - Output: (word, group of frequencies)

3. **Combine**:
   - Aggregates values for each key.
   - Example:
     - Input: (word, group of frequencies)
     - Output: (word, sum of frequencies)

4. **Reduce**:
   - Outputs the final result.
   - Example:
     - Input: (word, sum of frequencies)
     - Output: (word, final frequency)

Example:

- Input: "A woodchuck would chuck a lot of wood if a woodchuck could chuck wood."
- Output:
  - (woodchuck, 2)
  - (wood, 1)
  - (a, 1)
  - (chuck, 2)
  - (could, 1)
  - (if, 1)
  - (lot, 1)
  - (of, 1)

Example Output:

- (word, frequency):
  - woodchuck: 2
  - wood: 1
  - a: 2
  - chuck: 2
  - could: 1
  - if: 1
  - lot: 1
  - of: 1
Why is this hard?

Failure is common

Even if each machine is available $p = 99.999\%$ of the time, a datacenter with $n = 100,000$ machines still encounters failures $(1-p^n) = 63\%$ of the time

Data skew causes unbalanced performance across cluster

Problems occur at scale
Assignment 1.1 is due 2/3
Assignment 1.2 is due 2/8
Assignment 1.3 is due 2/10
Sequential MapReduce

Master

Map Phase
- Map Task 0
- Map Task N-1

Reduce Phase
- Reduce Task 0
- Reduce Task K-1

Merge
Distributed MapReduce

Master

Map Phase
- Map Task 0
- Map Task N-1

Reduce Phase
- Reduce Task 0
- Reduce Task K-1

Merge